2009-04-01

Test Case Generation using Model Checking for Software Components Deployed into New Environments

Tonglaga Bao
tonglaga@hotmail.com

Michael D. Jones
mike.jones@byu.edu

Original Publication Citation

T. Bao and M. Jones, "Test Case Generation using Model Checking for Software Components Deployed into New Environments", IEEE International Conference on Software Testing, Verification, and Validation Workshops, pp. 57-66. Denver, Colorado, April 29.

Test Case Generation using Model Checking for Software Components Deployed into New Environments

Tonglaga Bao
Computer Science Department
Brigham Young University
Provo, UT, USA
tonga@cs.byu.edu

Michael D. Jones
Computer Science Department
Brigham Young University
Provo, UT, USA
jones@cs.byu.edu

Abstract

In this paper, we show how to generate test cases for a component deployed into a new software environment. This problem is important for software engineers who need to deploy a component into a new environment. Most existing model based testing approaches generate models from high level specifications. This leaves a semantic gap between the high level specification and the actual implementation. Furthermore, the high level specification often needs to be manually translated into a model, which is a time consuming and error prone process. We propose generating the model automatically by abstracting the source code of the component using an under-approximating predicate abstraction scheme and leaving the environment concrete. Test cases are generated by iteratively executing the entire system and storing the border states between the component and the environment. A model checker is used in the component to explore non-deterministic behaviors of the component due to the concurrency or data abstraction. The environment is symbolically simulated to detect refinement conditions. Assuming the run time environment is able to do symbolic execution and that the run time environment has a single unique response to a given input, we prove that our approach can generate test cases that have complete coverage of the component when the proposed algorithm terminates. When the algorithm does not terminate, the abstract-concrete model can be refined iteratively to generate additional test cases. Test cases generated from this abstract-concrete model can be used to check whether a new environment is compatible with the existing component.

1 Introduction

As the complexity of software increases, so does the cost to develop, test, and maintain software. Component based software development decreases this cost by reusing software that has already been developed and tested. One difficulty of reusing a component is ensuring that a component developed in one environment works as expected in another environment.

In this paper, we address the problem of testing a new software environment for compatibility with an existing component. We define components as a reusable piece of code that can accomplish a certain task and is designed to interact with other components or programs. The environment is the rest of the software which closes the component to make it executable. The environment can include the other components or programs that interact with the component. A new environment is an environment other than the one in which the component is originally deployed. In this work, we assume the component is already defined and we do not address the problem of extracting a component from a software artifact.

This problem is important to software engineers who must deploy a component inside a new software environment. Since software is increasingly built from the existing components, verifying whether the already developed components work as expected in new environments become more important.

Existing techniques to address the problem of verifying software components include testing and formal verification. Software testing is useful in finding many errors in the early stages of software development. Since it is well understood that software can not be tested completely, finding a suitable set of test cases is critical in software testing. Model based testing is often used to generate a promising set of test cases by creating an abstract test model of the components to guide test case generation. The abstract model should capture important functionalities of the System Under Test (SUT) and result in a useful test suite. Model based testing technique generates test cases according to the test goal, such as a coverage criterion.

978-0-7695-3671-2/09 $25.00 © 2009 IEEE
DOI 10.1109/ICSTW.2009.15

The two main challenges for model based testing are how to generate a model and how to generate the test cases. The current approaches often generate the model manually. There are several drawbacks for this approach. First, It is a time consuming and error prone process to manually generate an abstract model of the SUT. In most cases, the SUT is too complex to be accurately modeled. Second, if the model is manually generated, generally there is a semantic gap between the model and the original system. Therefore, it might not be feasible to turn the abstract test cases into executable test cases. Third, once the model is generated, it is fixed. Refining the model to generate more test cases or adding new features to the original system requires an entire new modeling process. Testing an existing component within a new environment is difficult when using model based testing approaches because the new environment is often too complex to be modeled or there is simply no high level specification or source code available for the environment from which a model can be extracted.

Model checking complements software testing by locating errors which are difficult to find using software testing alone. However, applying model checking to a component is difficult due to the complexity of both the component and its environment.One technique is to abstract the environment and leave the component under test concrete. The abstract environment provides different inputs to the component so that the model checker can explore all behaviors of the component. The drawback of this approach is that abstracting the environment is difficult or impossible when there is no formal model for the environment. Furthermore, the component itself can be so complex that model checking it without abstraction is also impossible.

In this paper, we propose an original approach to generate test cases to verify the compatibility of existing components with new software environments. Our approach is a novel integration of model checking with model based testing to generate test cases for the new environment. Through the integration, we obtain the advantages of both model checking and model based testing and reduce the drawbacks of both techniques.

Given a component and the original environment, we abstract the component and leave the environment concrete. We start by executing the entire system normally. When program control enters the component, a model checker generates concrete system states from the component source code. We then apply an under-approximating predicate abstraction based on [14] to the concrete states. The abstraction saves time and space by reducing the model. When program control reaches the environment, we pause the model checker and save the program state into the test input set. We then initialize the environment using the program state and execute the system in the native execution environment. When program control returns to the component, we save the current program state into the test output set and resume the model checker. We repeat this process to obtain a set of input and output values. These values are test cases which can be used to test a new software environment for compatibility with the component.

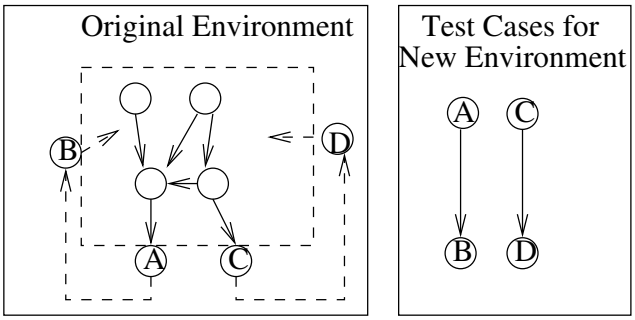


Figure 1. Test generation for a new environment using an abstract model of the component under test.

The abstraction approach has been used to model check software consisting of hundreds of lines of code in several medium sized C programs with several thousand lines of code in less than 5 minutes. These programs include open source software and student projects from an operating systems class. The results are discussed along with the implementation of the algorithm in SPIN in [1]. In this paper, we propose an adaptation to the method to support test generation for components deployed into new environments.

Figure 1 illustrates this approach. The left side of the figure shows the model composed of an abstract component, which is represented by the dotted square, and a concrete environment, which is represented by the solid square. The circles and arrows indicate program states and transitions respectively. The state space of the abstract component is explored until execution exits the component and enters the first environment state A. Then the program executes normally until it reaches the last environment state B and reenters the component. A similar process occurs for C and D. The right figure shows the resulting list of test cases for the new environment. In this work, we assume the environment has a single unique response for a given input.

In order to generate test cases using an abstract model of a component within a concrete model of the environment, we need to address two problems. One is to simplify communication between abstract components and concrete environments. The other one is to obtain a precise model for the abstract component through refinement. The precise model is a model that is bisimilar to the concrete component so that it behaves the same as the concrete component. Communication between component and environment

is simple when state exploration in both the component and environment is based on executing the original program on concrete states. If we use an abstract state in the component, however, all concrete states represented by the abstract state at the component interface must be generated, passed to the environment, executed in the environment and then recollected to form possibly new abstract states. We discuss completeness, correctness and refinement for an abstraction scheme based on [14] which address the above problems.

Generating test cases with the abstract component is possible because the new environment needs to have the same interface as the old environment in order for the component to function inside the new environment. Assuming there is a single unique response for any given input to the environment, test cases generated using an abstract model of the component are compatible with the new environment if the component is compatible with the new environment.

The goals of our test cases include getting a complete coverage of the component interface behaviors in the original environment and detecting errors in the new environment. The test coverage obtained by our approach depends on the refinement and termination of the algorithm. Assuming the run time environment is able to do symbolic execution either through instrumentation or some other mechanisms, we can obtain complete test coverage for the component interface in the original environment when our algorithm terminates. We also can obtain more behavioral coverage for the original environment by including a large part of the environment in the component. When the new environment fails the tests, we know there are errors in the new environment. If the new environment passes all the tests, then we can not claim anything about the new environment because the new environment might include more behaviors than the original environment. In this work we do not consider any real-time values or constraints in the environment under test. The complexity in obtaining a full path coverage through test generation using model checking is PSPACE-complete in the general case, however, in practice we notice a lower complexity.

Compared with traditional model based testing, the advantage of our approach is that both modeling and test generation are automatic. And since test cases are obtained through running the component within the original environment, they can be applied directly to the new environment without the need to transform abstract test cases into executable test cases. Furthermore, our modeling and test generation approach can be refined iteratively to obtain an increasingly precise model yielding more test cases.

This paper is organized as follows. In section 4.2, we present related work. In section 4.3, we give the definition of the model which we use to reason about. In section 4.4, we describe the algorithm we propose. In section 4.5, we establish the necessary conditions for a bisimulation between the abstract component model and the original software. In section 4.6, we conclude and discuss future work.

2 Related Work

We focus on related work in the areas of model based testing, model checking component based software, generating models through abstraction, and combining symbolic simulation with concrete exectuion.

Most of existing approaches to apply model based testing to the SUT generate models from high level specifications [4, 7]. The approach has the advantage of finding errors faster in the early stages of the development process. However, there is often a semantic gap between the high level specification and the actual implementation. Furthermore, if the high level specification is not written in a standard modeling language, a manual translation from the specification to the actual model is required, which is a time consuming and error prone process. Our method extracts a model directly and automatically from the source code to obtain an accurate model of the SUT fast. Our SUT is unique in a way that it is a mixed model of abstract component and concrete environment.

Existing work in model checking of component based software includes high level specifications of components as models and verifying concrete source code for a component inside an abstract environment. In [11, 2], the components are modeled using behavioral protocols. The drawback of this approach is that a precise high-level model of a component can be difficult to obtain. We obtain the component model by executing source code.

In [3], a program is divided into two parts: the component under test and the environment. The component under test is verified concretely inside the abstract environment which provides the necessary behaviors to close the environment. The drawback of this approach is it is a time consuming and error prone task to abstract the environment and the component itself may be too complex to be verified without an abstraction. In our approach, we abstract the component, which allows us to reason about more complex components and we leave the environment concrete which allows our environment to include language features such as complex data structures, pointers and library calls.

We apply Pasareanu et. al's [14] under-approximation abstraction scheme to the component under test. This abstraction also explores the concrete state space and stores concrete states into a queue or a stack. A set of predicates is used to abstract each concrete state. Each predicate is represented by a single bit in the abstract state. Precision in the abstraction is lost when two concrete states map to the same abstract state, but have different behaviors in the original system. An iterative refinement is done to include more and more behaviors of the original system by adding pred-

icates to the abstraction. The algorithm terminates when the resulting abstract state space is bisimilar to the concrete state space. Termination of the refinement process is detected by checking the weakest precondition of each transition in terms of the given predicates with the help of a theorem prover. In sections 3 and 5 we reason about similar properties of Pasareanu's abstraction scheme in a mixed concrete/abstract computational model.

There are several works which combine concrete execution with symbolic simulation to generate test cases. The concolic testing approach [15] starts by executing the unit under test with random inputs. Path constraints are then collected along with concrete execution paths. These path constraints are used to provide new inputs to the unit under test which drive concrete execution through an alternative path. We also combine concrete execution with symbolic simulation. Our approach is different in two major ways. First, concolic execution tests a unit by providing random input values. We test a new environment to see if it is compatible with an existing component by providing the environment the input that is generated by the component. Second, concolic execution uses symbolic execution and path constraints to generate new concrete input to the unit under test. We use symbolic simulationa and path constraint to generate new predicates which refine the component and this, in turn, generates new input to the environment. Compared with concolic testing, our approach is more restricted with theorem prover capabilities but gets better test coverage.

Pasareanu et al [13] extend JPF to generate test cases for the unit under test. They symbolically simulate the unit under test and concretely run the surrounding environment to drive the unit under test. Like our approach, it generates test cases by exhaustively exploring the state space of the components under test with the help of model checker in a concretely executed environment. The difference is [13] generates test cases by symbolically analyzing the unit under test, but we generate test cases by concretely executing the component under test.

3 Computational Model

In this section we present a computational model for abstract components inside concrete software environments for use in model based test generation. As mentioned earlier, we apply under approximated predicate abstraction based on [14] to model the components. Under approximating abstraction with predicates allows the detection of missed program behaviors through the use of weakest preconditions as described by Pasareanu et.al [14]. Later, we will describe how to reason about abstraction using weakest preconditions in our component representation.

Component A component is a reusable piece of code that can accomplish a certain task and which is designed to interact with other components or programs. We assume the source code of the component is available. In this work, we simply define a component as a set of program counter (pc) values. We let PC_c denote the set of pc values that belong to the component under test. The problem of extracting a component from a software artifact is important, but left as future work.

Our model of a system consists of a concrete part and an abstract part. We model both parts separately and then combine them in a mixed model as follows.

Concrete Model Given a finite set of atomic propositions AP, the state space of a program is modeled as a transition system $M = (S, s_0, T, L)$ where:

- S is a finite set of states,
- $s_0 \in S$ is the initial state,
- $T \subseteq S \times S$ is a set of transitions,
- $L : S \rightarrow 2^{AP}$ is a labeling function, where $L(s) = \{p \in AP \mid s \models p\}$. Here $s \models p$ means atomic proposition p is true in state s.

Figure 2 shows an example of the concrete model with other models that will be introduced later. In (a), several simple C program statements are given, and in (b) the corresponding concrete model is shown. Each circle represents a state and each arrow represents a transition. The first number in the circle represents the value of variable i and the second number represents the value of variable j.

Abstract Model (adapted from [14]) Assume $\Phi = \{\phi_1, \ldots, \phi_n\}$ is a set of predicates. Assume $\alpha_\Phi : S \rightarrow B_n$ is an abstraction function in which $B_n = \{0,1\}^n$ is a set of bit vectors where n is the number of predicates, $b_1 \ldots b_n \in B_n$ is a bit vector, and $\alpha_\Phi(s) = b_1 \ldots b_n$ with $b_i = 1$ if $s \models \phi_i$, else $b_i = 0$. Here $s \models \phi_i$ means the predicate ϕ_i is true in state s.

Given a concrete program model $M = (S, s_0, T, L)$, a set of predicates Φ, and an abstraction function α_Φ, the abstract model $A = (S_\alpha, a_0, T_\alpha, L_\alpha)$ where:

- $S_\alpha = \{a \mid s \in S \text{ and } a = \alpha_\Phi(s)\}$ is a set of abstract states,
- $a_0 = \alpha_\Phi(s_0)$ is the initial abstract state.
- $T_\alpha \subseteq S_\alpha \times S_\alpha$ is a set of abstract transitions.
- $L_\alpha : S_\alpha \rightarrow 2^{AP}$ is a labeling function for abstract states, where $L_\alpha(a) = \{p \in AP \mid a \models p\}$. and assume $AP \in \Phi$.

Figure 2 (c) shows the corresponding abstract model of (a) with dotted circles to represent abstract states. In this figure, we use two predicates $i < 10$ and $j < 10$ to abstract the system. The first boolean variable corresponds to $i < 10$ and the second boolean variable corresponds to $j < 10$. As is typical in predicate abstraction, we never abstract the program counter.

Mixed Model The mixed model includes both the abstract and concrete parts of the program. Given a concrete program model $M = (S, s_0, T, L)$, a predicate set Φ, and an abstraction function α_Φ, the mixed model $X = \{S_x, x_0, T_x, L_x\}$ is

- $S_x = \{x \mid x = f(s) \text{ and } s \in S\}$ is a mixed set of abstract and concrete states, where f is a function such that
$$f(s) = \begin{cases} s & pc \notin PC_c \\ \alpha_\Phi(s) & otherwise \end{cases}$$
If the pc value in the current state does not belong to the component, then the state remains concrete. Otherwise, the state gets abstracted.
- x_0 is a start state, so that if $pc \in PC_c$ then $x_0 = \alpha_\Phi(s_0)$, else $x_0 = s_0$,
- $T_x \subseteq S_x \times S_x$ is a set of transitions. More specifically, it consists of concrete transitions $T_x^{c\to c}$, abstract transitions $T_x^{\alpha\to\alpha}$, transitions between concrete and abstract states $T_x^{c\to\alpha}$, and transitions between abstract and concrete states $T_x^{\alpha\to c}$,
- $L_x : S_x \to 2^{AP}$ is a labeling function, where $L_x(x) = \{p \in AP \mid x \models p\}$

Figure 2 (d) represents the mixed model. We assume that program location 1 and program location 5 belong to the component and that the rest belong to the environment.

We write $s \xrightarrow{t} s'$ to indicate that there is a transition t between states s and s'. We write $s \to s'$ to denote a transition between s and s' when the context is obvious. If state s reaches s' through zero or more transitions, we write $s \to^* s'$ and say s' is reachable from s.

Path A path $\pi = s_0 \dots s_n$ in a mixed model is a finite sequence of states such that $(s_i, s_{i+1}) \in T_x$ for all $i \in 0 \dots n-1$. We use $\pi_{s_i}^{s_n}$ to denote that there is a path between s_i and s_n. Paths which start in the abstract component, enter the concrete environment and return to the abstract component will need special treatment later when defining verification properties. We call them border path. For example, in Figure 2(d), these five states compose a border path. we use $a_1 \xrightarrow{\alpha c^+ \alpha} a_2$ to indicate border path where $\xrightarrow{\alpha c^+ \alpha}$ represents the path starts with an abstract state, goes through one or more concrete states and ends at an abstract state.

For two paths $\pi_{x_0}^{x_n}$ and $\pi_{y_0}^{y_n}$, $\pi_{x_0}^{x_n} = \pi_{y_0}^{y_n}$ indicates that these two paths follow the same transitions, and $\pi_{x_0}^{x_n} \neq \pi_{y_0}^{y_n}$ indicates that one or more transitions in each path are different.

Under approximation abstracts away some behaviors of the concrete system. If there is no loss of precision in the behavioral model due to the abstraction for each transition relation, we say the abstraction is exact. Checking the exactness of the abstraction of the transition relation is the biggest challenge we face using mixed abstract and concrete models. If each transition between abstract states is exact with respect to the corresponding transition between concrete states, then a bisimulation relation can be established between the abstract and concrete models and CTL* properties will be preserved in the abstract model. When the transition resides entirely in the abstract part of the system, this check is done easily using weakest precondition [14].

However, the check is more complicated when we must check the exactness of a border transition. In order to explore all possible behaviors in such border cases, we need to reason about the accumulated effect of the intermediate transitions on the border states. We will use a method similar to symbolic simulation to capture the meaning rather than just the effect of concrete transitions on the border states.

Symbolic transitions A symbolic transition is an accumulation of the effects of a sequence of transitions on the state variables. For example, if we have a series of transitions $x = x + 1$, $x = 2x$, and $x = x^2$ in a path, then symbolic transition for these transitions is $x = (2(x+1))^2$. Given a path $\pi_{x_0}^{x_n}$, the notation (x_0, x_n) denotes the symbolic transition from x_0 to x_n. In other words,

$$(x_0, x_n) = (x_0, x_1) \circ (x_1, x_2) \circ \dots \circ (x_{n-1}, x_n)$$

where

$$0 \leq i \leq n-1 \text{ and } (x_i, x_{i+1}) \in T_x$$

and transition composition, $\circ$, denotes the sequential application of a transition to the result of the previous transition.

As in [14], the precision check is based on weakest preconditions. We must, however, cope with weakest preconditions defined over border transitions as well as component transitions. In either case, the basic definition is the same.

Weakest Precondition The weakest precondition, in terms of a transition and predicate ϕ, calculates a predicate that

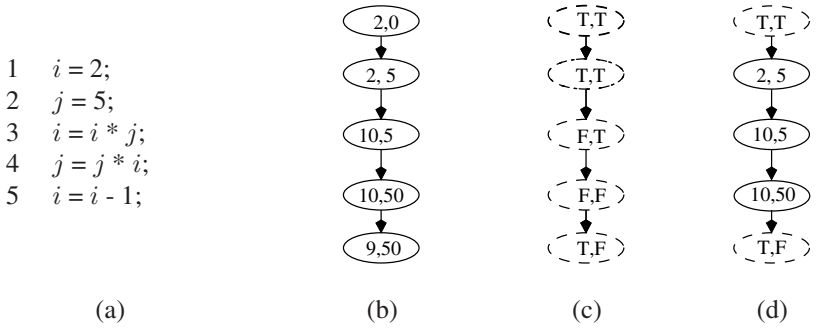


Figure 2. (a) Sample C program (b) Concrete system (c) Abstract system using predicates $i < 10$ and $j < 10$ (d) Mixed system

must be satisfied before a transition executes so that ϕ is satisfied after the transition. We use $wp(\phi, i)$ to express weakest precondition of transition i in terms of a predicate ϕ.

The $wp(\phi, i)$ = $\phi[x_{new}/x_{old}]$ where x_{new} is the new value of variable x in the predicate ϕ after a transition i, and x_{old} is the old value of variable x before the transition i.

4 Algorithm

In this section we describe the state enumeration algorithm for generating states and test cases in abstract components in the context of a concrete environment. We have omitted property checking and have assumed that all predicates are global in order to simplify the presentation.

Figure 3 shows the algorithm. It starts by calling the procedure **init** which takes a program *prog* as input. In line 2, Φ is initialized with all of the guards in the program. When a component is abstracted through predicate abstraction, the predicates in Φ are the initial set of predicates used to create abstract states during the first iteration of the algorithm. Φ_{new} stores the new predicates to refine the abstraction after each iteration and is initialized to the empty set in line 3. The **environment** function returns the first state which lies in the component.

When we have a start state that lies in the component, we push it on the stack at line 8 and call the **component** function in line 9. When using predicate abstraction with refinement, we repeatedly run the entire program until no refinement is necessary, as shown in line 10.

The **component** function explores the state space by storing abstract states in the hash table, storing concrete states on the stack, and executing transitions which leave the component without storing states. If the next transition exits the component, then we store the current state in the input set at line 20 and execute instructions in the environment until the program control returns to the component at line 21. The next state is generated by applying the current transition to the current state, line 22, or in the **environment** function at line 28. State exploration then continues by pushing the next state into the stack in line 23.

In the **environment** function, when the next instruction is in the environment, we simply execute it at line 28. Otherwise, we store the first state that lies in the component as the expected test output at line 31 and return it at line 33. Symbolic simulation of the border transition is performed in this function. The symbolic execution is used to refine the border transition. The test output and test input are paired and added to the test set at line 32.

5 Theorems

In this section, we analyze the mixed model we defined in section 2.3. First we discuss how to check the precision of the abstraction used in the component. Then we show how the under approximation abstraction scheme can be used to find test cases. After that, we discuss termination detection and property preservation in the mixed model.

5.1 Precision

The central problem in our mixed computational model is determining whether or not the abstraction is precise, or, in other words, creates a bisimulation between the partially abstract and the original systems. And this problem is particularly difficult for border transitions which span the boundary between the abstracted component and concrete environment.

This section focuses on reasoning about precision in predicate abstraction with a theorem prover [14]. There are two ways in which the mixed model can lose precision due to abstraction in the component. We illustrate each case

```
1   proc init(prog)
2     Φ := Guards(prog)
3     Φ_new := ∅
4     do
5       Φ := Φ ∪ Φ_new
6       if start_instr ∈ environment then
7         start_state = environment(start_instr, start_state)
8       push(start_state)
9       component()
10    while Φ_new ⊈ Φ
11
12  proc component()
13    while size(stack) != 0
14        cur_state = top(stack)
15        α = abstract (cur_state)
16        if (α ∉ hash table)
17          insert α into hash table
18          cur_inst = transition(cur_state)
19          if (cur_inst ∉ comp)
20            insert next_state into input
21            next_state = environment (cur_inst, cur_state)
22          else next_state = cur_inst(cur_state)
23          push (next_state)
24        else pop(stack)
25
26  proc environment(inst, state)
27    do
28      next_state = inst(state)
29      inst = transition (next_state)
30    while (inst ∈ environment)
31    insert next_state into output
32    insert (input, output) to test set
33    return (inst(next_state))
```

Figure 3. State enumeration algorithm that combines under-approximation with concrete execution.

with a simple example then discuss how to detect and recover lost precision. Proofs are given for each precision-recovery technique.

Figure 4 shows the first case. In this case, precision is lost because two concrete states are abstracted into the same abstract state, but exhibit different behaviors when they go through the concrete environment and return to the component. In (a), a simple C program is given. Here, we assume program locations 1 and 5 are in the component under test, and that program locations $2 \rightarrow 4$ are in the environment. We also assume that the initial value of i is 2, and that the initial predicate for abstraction is $i \geq 1$. Graph (b) represents the corresponding system. Solid circles represent concrete states and dotted circles represent abstract states. In each state, a small letter represents the name of the concrete state, and the number gives the value of i in that state. Capitol letters denote abstract states.

In start state a, i has the initial value of 2. Since program location 1 is in the component, we abstract the current state to abstract state A. Then program control enters the concrete environment. At this point, we execute the program through states b, c, and d until program control returns to the component at program location 5. Now in the current state e, the value of i becomes 1, which also satisfies the predicate. Therefore, state e is abstracted into abstract state C since PC values make it different than a. Next, program control returns to the beginning of the loop. Since the current state f has the same program counter value as state a and satisfies the same predicate, state exploration will stop at f. However, if we had allowed the concrete execution of f, then we would have reached state j, in which the value of i becomes 0. Since this state does not satisfy the predicate, it is abstracted into a different abstract state B, and this behavior will be missed.

To explore both state e and j, a refinement is needed to differentiate state a and f. We perform refinement by checking the weakest precondition of the accumulated effect of the concrete transitions in the border path. In this example, we will take the accumulated behaviors of $i = i+3, i = i-1, i = i-2, i = i-1$, that is $i = i+3-1-2-1$, which will be $i = i - 1$. Then we check if the current predicate implies the weakest precondition of this accumulated transition. In other words, we check if $i \geq 1 \Rightarrow$ wp($i \geq 1$, $i = i - 1$), that is equivalent to checking if $i \geq 1 \Rightarrow i-1 \geq 1$. Since this is false, we will add $i-1 \geq 1$ to the predicate set to refine the system. Now states a and f are differentiated under the abstraction.

For simplicity, given a set of predicates Φ and a concrete state s, we sometimes write $\alpha_\Phi(s)$ to denote the conjunction of all predicates in the set Φ which evaluate to true together with the negation of remaining predicates in Φ which evaluate to false for state s.

Theorem 1 states that doing abstraction refinement on

symbolic representations of transition sequences preserves the behavior of paths through boundary states.

Theorem 1. *: Let Φ denote the set of predicates to abstract the system. Given two paths $\pi_{x_0}^{x_n}$ and $\pi_{y_0}^{y_n}$, such that $\pi_{x_0}^{x_n} = \pi_{y_0}^{y_n}$, $\alpha_\Phi(x_0) = \alpha_\Phi(y_0)$, but $\alpha_\Phi(x_n) \neq \alpha_\Phi(y_n)$, let $\Phi' = \Phi \cup \{wp(\alpha_\Phi(x_n), (x_0, x_n))\}$ then $\alpha_{\Phi'}(x_0) \neq \alpha_{\Phi'}(y_0)$*

Proof. Doing symbolic simulation with the transitions in the border path is equivalent to checking weakest precondition of the transitions in the border path in reverse order. since $\alpha_\Phi(x_n) \neq \alpha_\Phi(y_n)$, the $wp(\alpha_\Phi(x_n), (x_{n-1}, x_n))$ is a predicate that is able to differentiate state x_{n-1} and y_{n-1}, which is also equivalent to one step of symbolic simulation according to the definition of the weakest precondition. Then, this weakest precondition is symbolically substituted to produce the next level of weakest precondition following the current transition. This newly produced weakest precondition can differentiate state x_{n-2} and y_{n-2} and so on. This is repeated until we obtain a predicate which differentiates x_0 and y_0.

□

The second case in which precision is lost occurs when the execution path branches in the environment. If the execution path branches in the environment, then the branch guard is propagated to the predicate set.

This is shown in the graph in Figure 5. In (a), we assume program locations 1 and 6 are in the component and that the rest are in the environment. All other assumptions are the same as Figure 4.

In graph (b) of Figure 5, border states a and g pass through the same transitions $i = i + 3$ and $i = i - 1$, reach states c and m, then separate into different transitions, and result in different abstract states C and B when they re-enter the component. As in the above example, g is abstracted into the same abstract state after the while loop iterates once. The state space exploration algorithm will backtrack and miss the behavior of abstract state B.

The intuition of our approach is to push the guards that distinguish c and m upwards into the path to generate predicates that distinguish a and g. In this example, we use the predicate $i == 3$. First, we create a symbolic transition for the series of transitions before the branch. In this case it will be symbolically simulating $i = i + 3$ and $i = i - 1$, which will be $i = i+3-1$, which is simplified to $i = i+2$. Then we check if $i \geq 1 \Rightarrow$ wp $(i == 3, i = i + 2)$. This is equivalent to checking if $i \geq 1 \Rightarrow i + 2 == 3$. Since this implication does not hold, we add $i == 1$ to the predicate set to refine the abstract system in the next iteration. Now, state a and g will be abstracted into different abstract state because of the newly added predicate.

Theorem 2. *: Let Φ denote the set of predicates to abstract the system. Given two path $\pi_{x_0}^{x_n}$ and $\pi_{y_0}^{y_m}$, such that*

$\alpha_\Phi(x_0) = \alpha_\Phi(y_0), \alpha_\Phi(x_n) \neq \alpha_\Phi(y_m)$
$x_0 \rightarrow^* x_{i-1} \xrightarrow{t_x} x_i \rightarrow^* x_n, y_0 \rightarrow^* y_{i-1} \xrightarrow{t_y} y_j \rightarrow^* y_m,$
$t_x \neq t_y, \pi_{x_0}^{x_{i-1}} = \pi_{y_0}^{y_{i-1}}$, *and* $\pi_{x_0}^{x_n} \neq \pi_{y_0}^{y_m}$

let $\Phi' = \Phi \cup \{wp(\alpha_{\Phi''}(x_{i-1}), (x_0, x_{i-1}))\}$
in which Φ'' includes a set of guards that differentiate x_{i-1} and y_{i-1}, then
$\alpha_{\Phi'}(x_0) \neq \alpha_{\Phi'}(y_0)$.

Proof. Proof is similar to theorem 1. Since Φ'' includes a set of guards that distinguish x_{i-1} and y_{i-1}, we have that $\alpha_{\Phi''}(x_{i-1}) \neq \alpha_{\Phi''}(y_{i-1})$. We propagate this predicate up towards the path until we get a predicate that differentiates x_0 and y_0. □

The next theorem says that if the abstraction is exact in terms of the symbolically simulated transitions which lie outside the component, then the abstraction includes all the possible behaviors of the states at the border of the component.

Theorem 3. *Given a series of transitions $x_0 \rightarrow x_1 \rightarrow \ldots \rightarrow x_n$, if the abstraction is exact for a symbolically represented transition (x_0, x_n) using a set of predicates Φ, then the abstraction captures all effects of concrete behaviors starting at location 0 and ending at program location n in terms of the predicates Φ.*

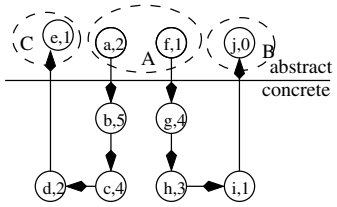
Proof. This theorem follows from theorem 1. Suppose $\alpha_\Phi(x_0)$ is an abstract state produced in program location 0, and $\alpha_\Phi(x_n)$ is an abstract state produced in program location n. The concrete transition $x_0 \rightarrow x_1 \rightarrow \ldots \rightarrow x_n$ is replaced by symbolic transition (x_0, x_n). States x_0 and x_n can be abstracted into existing abstract states or new abstract states. There are four different possibilities. First, both $\alpha_\Phi(x_0)$ and $\alpha_\Phi(x_n)$ are existing abstract states. Second, $\alpha_\Phi(x_0)$ is an existing abstract state and $\alpha_\Phi(x_n)$ is a new abstract state. Third, $\alpha_\Phi(x_0)$ is a new abstract state and $\alpha_\Phi(x_n)$ is an existing abstract state. Fourth, both $\alpha_\Phi(x_0)$ and $\alpha_\Phi(x_n)$ are new abstract states. In the first case, since both states are existing states, no precision is lost. The second case is impossible since we assume the abstraction is exact. In the third case, the new state is generated in program location 0 and the result of execution is an existing state at program location n, which does not lose precision. In the forth case, a new abstract state is generated in program location 0, which goes to a new abstract state at program location n, which also does not lose precision. Therefore, the theorem holds for all possible cases. □

5.2 Under-approximation

Checking precision for border states is easier if we consider only a prefix of the sequence of transitions in the

```
1  while (i ≥ 1){
2    i = i + 3
3    i = i − 1;
4    i = i − 2;
5    i = i − 1;
6  }
```

(a)

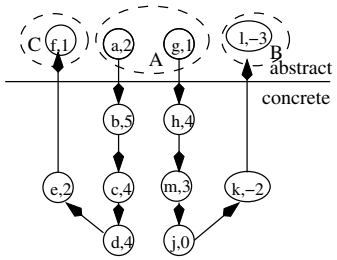


(b)

Figure 4. Loss of precision in a simple C program

```
1  while (i ≥ 1){
2    i = i + 3
3    i = i − 1;
4    if (i == 3) i = 0;
5    i = i − 2;
6    i = i − 1;
7  }
```

(a)



(b)

Figure 5. Loss of precision in a C program with a branch

concrete environment because there are less transitions to consider in the symbolic simulation. The precision of the component abstraction will increase as the number of concrete transitions considered increases. When we consider all the transitions, and the verification algorithm terminates, we have an exact abstraction which creates a bisimulation between the program and its abstraction. The following theorem precisely states these claims.

The next theorem states we can vary the precision of the abstraction by varying the number of concrete transitions used to build the symbolic representation of the surrounding software.

Theorem 4. *Suppose* $x_0 \overset{\alpha c^+ \alpha}{\rightarrow} x_n$*, then the abstract component generated by doing abstraction refinement in terms of* (x_0, x_i) *for the predicate set* Φ_i*, with* $0 < i < n$*, is an under-approximation of the original component.*

Proof. All states and transitions visited during model checking are concrete states and generated using concrete transitions from concrete software. Since $0 < i < n$, the exactness check will include concrete behaviors up to the instruction at program location i. However, program behaviors between the instructions at locations i and n are ignored in the precision check. This means that some execution paths that split between locations i and n may be missed. Therefore, the abstraction scheme we use results in an under-approximation and we can increase precision by including more of the environment in the component. □

If we treat each of the border transitions as one single transition by symbolically simulating it, then we will get a system that is presented in [14] and therefore we get a bisimilar system when the algorithm terminates as shown in [14]. Iteratively refining and symbolically simulating more of the SUT increases the precision of the model and supports the generation of increasingly complete sets of test cases.

5.3 Termination and Property Preservation

Termination is difficult to detect when execution leaves the component and never returns. This can occur when the program enters an infinite loop in the environment or when the program actually terminates in the environment. We can not, in general, detect the first case and we simply add an "exit" marker to identify the second case.

6 Conclusion and Future Work

We have presented an approach to generating test cases for a new environment into which an existing component will be deployed. Our approach is novel integration of model based testing with model checking and abstraction.

We automatically obtain a mixed model of an abstract component within a concrete environment from the component source code and the environment in which it is currently deployed. A relatively complete set of test cases are generated from this model utilizing the ability of model checker to exhaustively explore a system. The model also can be iteratively refined by an abstraction refinement schemes to provide more complete test cases.

Avenues for future work include evaluating the idea in real world software and automatic methods for extracting components.

References

[1] Tonglaga Bao and Mike Jones. Model checking abstract components within concrete software environments. In 15th International SPIN Workshop on Model Checking of Software (SPIN), Los Angeles, CA, 2008.

[2] Tomas Barros, Ludovic Henrio, and Eric Madelaine. Verification of distributed hierarchical components. In International Workshop on Formal Aspects of Component Software (FACS), Macao, China, 2005.

[3] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In 22nd International Conference on Software Engineering (ICSE), pages 439–448, Limerick, Ireland, 2000.

[4] Ian Craggs, Manolis Sardis, and Thierry Heuillard. Agedis case studies: Model-based testing in industry. In 1st European Conference on Model-Driven Software Engineering, Nuremberg, Germany, 2003.

[5] Peter C. Dillinger and Panagiotis Manolios. Enhanced probabilistic verification with 3spin and 3murphi. In 12th International SPIN Workshop on Model Checking of Software (SPIN), pages 272–276, San Francisco, CA, 2005.

[6] Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. Model Checking, pages 146–154. The MIT Press, 2002.

[7] G. Friedman, A. Hartman, K. Nagin, and T. Shiran. Projected state machine coverage for software testing. In Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis (ISSTA), pages 134–143, New York, NY, 2002.

[8] G. J. Holzmann. An analysis of bitstate hashing. In Proc. 15th Int. Conf on Protocol Specification, Testing, and Verification, pages 301–314, 1995.

[9] Gerard J. Holzmann and Rajeev Joshi. Model-driven software verification. In 11th International SPIN Workshop on Model Checking of Software (SPIN), pages 76–91, Barcelona, Spain, 2004.

[10] Gerard J. Holzmann and Margaret H. Smith. Feaver 1.0 user guide.

[11] Pavel Jezek, Jan Kofron, and Frantisek Plasil. Model checking of component behavior specification: A real life experience. In International Workshop on Formal Aspects of Component Software (FACS), Macao, China, 2005.

[12] Dritan Kudra and Eric G. Mercer. Finding termination and time improvement in predicate abstraction with under-approximation and abstract matching. In (MS thesis, Brigham Young University), 2007.

[13] Corina S. Pasareanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael R. Lowry, Suzette Person, and Mark Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In ISSTA, pages 15–26, Seattle, WA, 2008.

[14] Corina. S. Pasareanu, Radek Pelanek, and Willem Visser. Concrete model checking with abstract matching and refinement. In 17th International Conference on Computer Aided Verification (CAV), pages 52–66, Edinburgh, Scotland, 2005.

[15] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, pages 263–272, New York, NY, USA, 2005.